ECE 250 Project 2 Hashtables
Design Document

Anthony Xu, UW User ID:a68xu

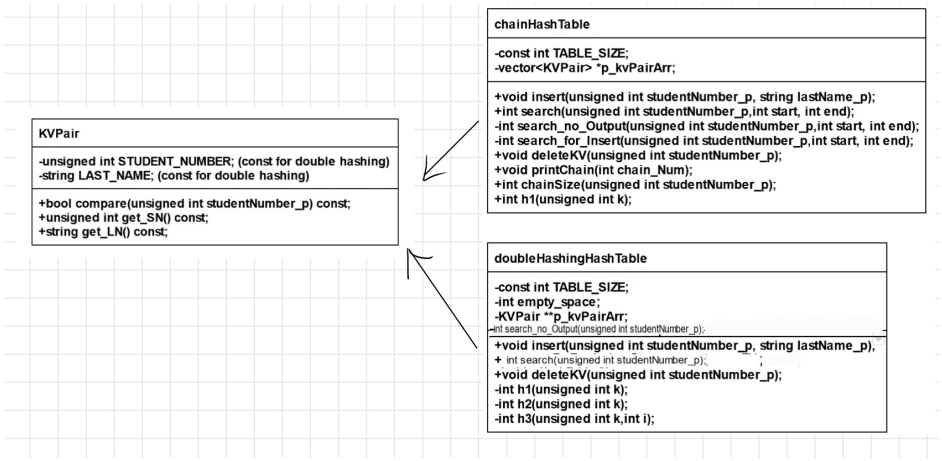
Nov 07 2022

Overview of classes:

Classes used:

- KVPair
  - KV stands for key value, each KVPair holds a key(student number) and a value(name).
- doubleHashingHashTable
  - The hash table data structure for double hashing, which uses a dynamic array of “pointers to KVPairs”. Supports the required operations in the project document. Allows double hashing
- chainingHashTable
  - The hash table data structure for chaining, contains a dynamic array. Each index of the array holds a vector of KVPairs. Supports the required operations in the project document, allows chaining.

UML Class diagram



From: http://www.cs.utsa.edu/~cs3443/uml/uml.html

Details on design decisions

KVPair Class

```
class KVPair
{

private:
    unsigned int STUDENT_NUMBER;
    string LAST_NAME;

public:
    KVPair(unsigned int studentNumber_p, string lastName_p);
    ~KVPair();
    bool compare(unsigned int studentNumber_p) const;
    unsigned int get_SN() const;
    string get_LN() const;
};
```

Member Variables

- The key and value of the KVPair is const for double hashing, since we shouldn’t modify them after initializing them.
- But no const for chaining, since when using chaining, the empty vectors would automatically assign default values when initializing the array.

Member Functions

- The constructor initializes the two private member variables.
- Compare allows for easy comparison so we know if we found the KVPair when deleting or searching. const due to no modification
- Getters and setters to access member variables.

int search_no_Output(unsigned int studentNumber_p);
int search(unsigned int studentNumber_p);

doubleHashingHashTable Class

```
class doubleHashingHashTable{
    private:
    const int TABLE_SIZE;
    int empty_space;
    KVPair **p_kvPairArr;
    int search_no_Output(unsigned int studentNumber_p);

    public:
    doubleHashingHashTable(int table_size_p);
    ~doubleHashingHashTable();
    void insert(unsigned int studentNumber_p, string lastName_p);
    int search(unsigned int studentNumber_p);
    void deleteKV(unsigned int studentNumber_p);
    int h1(unsigned int k);
    int h2(unsigned int k);
    int h3(unsigned int k,int i);

};
```

Member variables:

- TABLE_SIZE to hold the size of the hash table, should be constant throughout lifetime of the table
- Empty_space, so we know if the table is full or not, kept track of when we insert or delete each time. Allows for knowing if the table is full in constant time
- p_kvpairArr Pointer to the base address of the array of pointers to kv paris

Member Functions:

- Constructor, initialize the TABLE_SIZE using member initialization list. Initialize the array of pointers with an array size, which is the TABLE_SIZE
- Destructor, call delete on each index of the array of pointers, and then call delete on the array.
- Insert call h3 which is the hashing function that combines h1 and h2 for double hashing, check if the element is already there, insert if the slot is empty
- Search
  search by calling h3 and iterate for TABLE_SIZE times to see if slot h3() has the target student number
  If no matches after iterating TABLE_SIZE times, then not found
  Returns the index where the target is found, or -1 if not found
  Search_no_output is for the deleteKV function, same as search but no output
- deleteKV, first call the search_no_output function to see if the target SN is in the hash table, then perform deletion by calling delete on the index and setting the index to null , or indicate failure

chainingHashTable Class

```
class chainingHashTable{
    private:
    const int TABLE_SIZE;
    vector<KVPair> *p_kvPairArr;
    int search_no_Output(unsigned int studentNumber_p,int start, int end);
    int search_for_Insert(unsigned int studentNumber_p,int start, int end);

    public:
    chainingHashTable(int table_size_p);
    ~chainingHashTable();
    void insert(unsigned int studentNumber_p, string lastName_p);
    int search(unsigned int studentNumber_p,int start, int end);
    void deleteKV(unsigned int studentNumber_p);
    void printChain(int chain_Num);
    int chainSize(unsigned int studentNumber_p);
    int h1(unsigned int k);
};
```

Member Variables:

- TABLE_SIZE which is const
- Pointer to the base address of array of vectors of KVPairs

Member Functions

- The constructor initializes TABLE_SIZE using member initialization list and then initializes the array with size TABLE_SIZE
- The destructor calls delete[] on the p_kvPairAr
- Insert
  first checks if the index through the h1 function is has an empty vector, if empty, then insert the SN as the first element of the array
  If the vector is not empty, examine the vector with binary search method to see if the SN is already there, if not, insert at the correct spot to insure the list is in descending order
- search first compute h1(studentNumber_p), then search the vector using binary search to see if the SN is in the vector. Return -1 if not found, return vector index if found
- Search_no_output, same as search but no output
- deleteKV first calls the search_no_output function to see if the SN is in the list, if in the list, perform the deletion with the index returned

Test Cases

Double Hashing

1. Test repeated insertion and deletion
2. Test repeated insertion, deletion and searching
3. Test insertion until full, searching, and continue insertion, deletion, and then insertion with printing for verification
4. Test insertion when full
5. Test deletion when table is empty, when table is full, when the key doesn't exist in the table
6. Test for small table size 1

Chaining

1. Test repeated insertion, deletion,searching and printing
2. Test insertion and deletion for elements that are at the end or start or middle of a specific chain
    a. Test for various sizes of the chain, empty chain, big chain, single number of elements chain, double number of elements chain, one single element chain
3. Try different list sizes such 5, 7, 8, 10, 9, 12, 13, 15, 16
4. Test for small table size 1

Run Time

doubleHashingHashTable

- Insert
    - Ω(1) If the initial slot computed by h3 is empty or if the slot is already occupied by the key that we want to insert, then the operation ends, takes constant time to compute h1 and verify if the slot is empty or occupied
      Tight bound
    - O(N)In the worst case, we might end up iterating through the whole table to perform checking if slot if empty, and if the SN in the slot is different then the one we are inserting, which takes O(N)
    - O(1) Average case, O(1)
- Search
    - Ω(1) if we can find the value in the slot computed by the first iteration of h3, then it takes constant time to find the key
    - O(N) if we are unlucky and have to traverse the whole table to locate the key, it takes O(N)
    - O(1) Average case, O(1)
- Delete
    - Ω(1) if we can find the value in the slot computed by the first iteration of h3, then it takes constant time to find and delete the key
    - O(N) if we are unlucky and have to traverse the whole table to locate the key, or to find that the key is not in the table, it takes O(N)
    - O(1) Average case, O(1)

chainingHashTable

- Insert
    - Ω(1) If the vector at the " index computed by h1" of the array is empty, or if our value is larger or smaller than all values in the vector, then the insertion takes constant time
        - O(1) to computed h1
        - O(1) to insert at head or tail of the vector
    - O(logN)In the worst case, we will end up with all keys being inserted at the same chain of the table, for this we need to search the chain using binary search to locate the index in which we should insert our value in
        - O(logN) to find where we should insert at in the vector
        - O(1) to perform insertion-
    - O(1) average case. If we assume uniform hashing, it is equally likely for each chain to receive a new KVPair. The load factor(average number of keys per slot) is N/m. Since m is proportional to N, N = O(m). O(m)/m = 1, and since we also search using binary search, it is even faster. Therefore it takes O(1) on average.
        - O(1) to computed h1, the hash function
        - O(logN/m) which is O(1) to find where to insert in the list
        - O(1) to perform insertion since we already know the index

  Search
    - Ω(1) If the vector at the " index computed by h1" of the array is empty, or if our value is larger or smaller than all values in the vector, then the insertion takes constant time
        - O(1) to computed h1
        - O(1) to insert at head or tail of the vector
    - O(logN)In the worst case, we will end up with all keys being inserted at the same chain of the table, for this we need to search the chain using binary search to locate the index in which we should insert our value in
        - O(logN) to find where we should insert at in the vector
        - O(1) to determine whether each index has our key.
    - O(1) average case. If we assume uniform hashing, it is equally likely for each chain to receive a new KVPair. The load factor(average number of keys per slot) is N/m. Since m is proportional to N, N = O(m). O(m)/m = 1, and since we also search using binary search, it is even faster. Therefore it takes O(1) on average.
        - O(1) to computed h1, the hash function
        - O(logN/m) which is O(1) to find where the key would be in the list
        - O(1) each time to compare target key with the keys in the hash table
- Delete
    - Ω(1) If the vector at the " index computed by h1" of the array is empty, or if our value is larger or smaller than all values in the vector, then the insertion takes constant time
        - O(1) to computed h1
        - O(1) to insert at head or tail of the vector
    - O(logN)In the worst case, we will end up with all keys being inserted at the same chain of the table, for this we need to search the chain using binary search to locate the index in which we should insert our value in
        - O(logN) to find where we should insert at in the vector
        - O(1) to determine whether each index has our key.
    - O(1) average case. If we assume uniform hashing, it is equally likely for each chain to receive a new KVPair. The load factor(average number of keys per slot) is N/m. Since m is proportional to N, N = O(m). O(m)/m = 1, and since we also search using binary search, it is even faster. Therefore it takes O(1) on average.
        - O(1) on average to perform searching to find where the key would be stored.
        - O(1) to delete the KVPair if it is not already deleted
- Print
    - O(N) if we have all keys in a single slot
    - O(N/m) which is O(1) on average if m is proportional to N, since we need to traverse the whole chain and access all keys,
        - Takes O(1) to print each key